DEFINING DOCUMENT STYLES FOR WYSIWYG PROCESSING

DONALD D. CHAMBERLIN, HELMUT F. HASSELMEIER, AND DIETER P. PARIS

IBM Almaden Research Center

ABSTRACT

Recent years have shown two distinct but converging trends in document processing: the trend toward direct manipulation, or "WYSIWYG" systems, and the trend toward high-level generic markup. The Quill project at IBM Research is an attempt to combine the flexibility and ease of use of a WYSIWYG interface with the formatting power of the international standard SGML markup language. The Quill system will present a WYSIWYG user interface but will format documents under control of an external Document Design that specifies the degree of user control over document appearance. Quill includes a tool called the Designer's Workbench that enables a Document Designer to specify the syntax and semantics of a given type of document. Each element in the document type is defined by a "look" consisting of a property sheet and an optional semantic routine. The semantic routines are written in a high-level programming language and can call a set of system-provided utility functions that are designed, according to rules described in this paper, to be suitable for WYSIWYG processing.

1. Introduction

Over the last decade, the field of computer-based document processing has encompassed two different types of systems, originating from different sources and serving different requirements, and following distinct but converging evolutionary paths. The first type of system, which we will label with Ben Schneiderman's term "direct manipulation" [Schn83], is based on providing the user with a set of tools for directly controlling the physical appearance of documents. The first direct manipulation systems were simple word processors, based on a typewriter-like document model, which allowed users to manipulate text on a display screen using monospaced fonts and fixed line spacing. Powerful microprocessors, bit-mapped displays, and inexpensive laser printers made possible a new generation of direct manipulation systems: the WYSIWYG, or "what you

see is what you get", systems, which provide the user with an accurate display of formatted pages with multiple fonts and sizes, often including graphics as well as text. WYSIWYG systems such as Interleaf [INT86] allow the user to specify the appearance of objects on the page by means of pop-up "property sheets" that control various properties such as fonts and justification. Some WYSIWYG systems allow users to give names to certain sets of properties and to invoke these names at various places in the document to create objects with uniform appearance. WYSIWYG systems combine editing with formatting in an interactive interface that immediately applies each editing change to the document. The WYSIWYG approach to document creation is characterized by simplicity, flexibility, and ease of use; it is typically applied to the "office" world of small and medium-size documents but is evolving toward larger and more complex documents and more sophisticated typographic features.

The other principal approach to document creation, which we will refer to as the "markup" approach, originated with the idea that processing commands could be imbedded in the text of a document and used to control the process of compiling the document into a form suitable for printing. The "compilation" process typically applies to a whole document at a time and takes place without user intervention. The first markup systems were based on procedural commands that directed the machine to take specific actions such as "skip a line" or "change fonts". More recent systems are based on the concept of "generic markup" pioneered by Brian Reid's Scribe system [Reid80], in which logical elements of a document such as lists, examples, and footnotes are identified by descriptive "tags", and their physical appearance is controlled by a "style definition" external to the document. The concept of generic markup, as exemplified by the Standard Generalized Markup Language (SGML) [ISO86], offers many advantages including device independence, enforcement of style standards, and use of a marked-up document for multiple purposes. Since generic-markup systems process documents according to an external style definition, the style definition can perform complex functions such as automatic numbering of figures, resolving of references, and generation of a table of contents. Therefore, markup-based systems tend to be characterized by sophisticated functions that often require more than one formatting pass over the document, and are typically used in the "publishing" world of relatively long and complex documents. Recent developments in markup-based systems point toward greater

interactivity, including soft-copy preview facilities that provide a display of the formatted document before printing, and systems such as ICEF2 (formerly called Janus) [Cham87a] that provide a means for a user to edit and reformat individual pages without processing the entire document.

Viewing the two classes of document creation systems in a historical perspective, it seems clear that they are evolving along converging paths. We believe that a need exists for a new generation of systems that combines the ease of use and flexibility of WYSIWYG systems with the formatting power and generic processing capability of markup-based systems. The Quill project at IBM's Almaden Research Center is an attempt to build an experimental prototype of such a system.

Any attempt to combine WYSIWYG processing with generic markup must confront the central question of who is "in charge" of document appearance, since traditionally WYSIWYG and markup-based systems have taken opposite approaches to this question. In Quill, the extent of user control over appearance becomes a policy decision rather than a built-in characteristic of the document processing system. The Quill user is provided with a WYSIWYG display of the formatted document. To insert a new object into a document, the user invokes the generic name of the object (e.g., "footnote"), which causes the object to be created according to an external style definition. However, the external definition (called the "look") of any given object can be as rigorous or as flexible as desired by the Document Designer, who is simply a user of a Quill facility called the Designer's Workbench. A Document Designer might choose to specify all the properties of an object completely, leaving the user no options other than to invoke the object by its name. Alternatively, the Document Designer might choose to specify a set of default properties for the object-type, allowing end users to override certain of these properties (e.g., fonts) for individual objects in individual documents. The Designer's Workbench allows the Document Designer to specify the properties of various objects by choosing options on Property Sheets; similarly, the Quill editor allows the author of an individual document to control the properties of individual objects by choosing options on Property Sheets that contain those properties specified as overrideable by the Document Designer. A style specification might permit all properties to be overridden by users, thus enabling Quill to be used with all the flexibility of a WYSIWYG system. A style specification for a different type of document might rigorously specify all properties in order to enforce the

style standards of an organization. Other style specifications might implement an intermediate approach, exposing some properties but not others, according to the policy of the Document Designer for specific types of documents or classes of users.

A prototype implementation of Quill is underway at Almaden Research Center, using IBM RT PC workstations and the experimental QuickSilver operating system and "Whim" window manager [Good85]. The principal characteristics of the Quill prototype are as follows:

- Quill is based on the document model of the Standard Generalized Markup Language [ISO86]: a document is a hierarchy of nested elements, and the semantics of an element may depend on its position in the hierarchy. Validity of document structure is enforced according to an external SGML Document Type Definition.
- Quill allows a user to edit a document by either logical or physical actions. An example of a physical action is a keystroke or other editing action applied directly to a WYSIWYG display. An example of a logical action is changing the markup of an element from "numbered list" to "bulleted list", which automatically reformats the list and affects the appearance of its items.
- Quill formats documents under the control of an external style definition that can specify complex processing, including remote effects such as resolving references and generating a table of contents. The style definition is created and edited by a facility called the Designer's Workbench.
- Quill allows a document to contain graphics and other materials as well as text, nested inside each other without restriction. Each type of material (e.g., graphics, mathematics, chemistry) can be manipulated by its own specialized editor, and the Quill system can be extended by the addition of new specialized editors. The representation of graphic objects using SGML, and the mechanisms used to support extensibility in Quill, have been described in other papers [Cham87b, Cham87c].

2. Elements, Attributes, and Properties

In the document model of the SGML Standard, a document consists of a hierarchical collection of *elements*. Each element is identified by a *tag*, which consists of a *generic identifier* (element name) and zero or more *attributes*, each of which has a name and a value. Each type of element

has its own set of attributes, which are defined in a formal *Document Type Definition.* In addition to its tag, an element may have *content* which consists of character data and other elements.

The SGML Standard specifies no fixed set of elements and attributes, but allows them to be defined by the creator of the Document Type Definition (DTD). Quill will conform to the SGML Standard in that the Quill Document Designer can specify any desired set of elements and attributes. During an editing session, the Quill user can view and edit the attribute values of any element in the document. In addition, the Quill user can view and edit the *properties* of an element; these are presented in the form of a Property Sheet like the one shown in Figure 1. The Property Sheet contains values for properties such as fonts, alignment, etc., that control how Quill formats the given element. By examining the Property Sheet, the user can see the values of the properties, and, *where authorized by the Document Designer,* can modify these values. Those properties whose values are not user-modifiable for a given element appear shaded on the property sheet. The user can change an unshaded property by pointing to its value with the mouse and clicking the mouse button; this causes a menu of appropriate values to appear, from which the user can select the desired value by another click. Where appropriate, units (e.g., inches, picas, millimeters) also appear on the Property Sheet and can be changed by mouse clicks. At the bottom of the Property Sheet are three buttons that allow the user to confirm or cancel the revised set of properties, or to reset all properties to their default values.

The Quill Designer's Workbench allows the Document Designer to view and edit the Property Sheet for any type of element in a given Design. The property sheets seen by a Designer are much like those seen by an end user while editing a document. The Designer can specify property values in either absolute form (e.g., "14 points") or relative form (e.g., "4 points larger than the current size"). The Designer can change the value of any property, and, in addition, can control which properties are "locked" against user overrides (clicking on the name of a property cycles it between "locked" and "unlocked"; "locked" properties appear shaded on the property sheet). The property values specified by the Document Designer become the default property values for the given element, subject to user overrides if so authorized.

During an editing session, Quill maintains a data structure called the "Logical Tree" which contains a node for each element in the document.

Figure 1: A Property Sheet

Page	Break:	YES			
	Width:	8.5 INCH		Height:	11 INCH
Margins:	Top:	1 INCH	Page Number:	Value:	CONTINUE
	Bottom:	1 INCH		Style:	ARABIC
	Left:	1 INCH		Horiz. Pos.:	3.2 INCH
	Right:	1 INCH		Vert. Pos.:	0.6 INCH
Section	Break:	YES			
	Number of Cols.:	2		Column Width:	3 INCH
Column	Break:	NO			
Shape	Line Break Top:	YES			
	Line Break Bottom:	YES		Sticky at Bottom:	NO
	Alignment:	JUSTIFIED		Left Indent:	0 INCH
	Top Space:	1 LINE		Right Indent:	0 INCH
	Bottom Space:	1 LINE		Hanging Indent:	3 P.P
	Keep:	NO		Widow Prevention:	YES
Tabs: 0.25 0.50 0.75 1.00 1.25 1.50 1.75 2.00 2.25 2.50 INCH					
Font	Type Family Name:	TIMES ROMAN			
	Point Size:	10 Pts		Weight:	MEDIUM
	Italics:	OFF		Width:	NORMAL
	Baseline Offset:	0 INCH		Line Space:	12 P.P

Confirm Cancel Reset

Each node points to a set of default properties for its element-type, as defined by the Document Design. In addition, each node in the Logical Tree records those properties, if any, which have been given specific values for the individual element represented by the node. Thus, when some editing change is made to the element (e.g., a keystroke causes a character to be added to its content), the applicable properties are readily accessible for reformatting the element. It is a key characteristic of WYSIWYG systems that any element in the document may be reformatted at any time due to user actions, and therefore each element must be represented by a data structure that contains all the information necessary for its reformatting.

At the conclusion of a Quill session, when the document is to be saved in an SGML file, it is necessary to represent user overrides to individual properties using SGML syntax. The SGML Standard provides a feature

called "Link Sets" for this purpose, which can be used to associate one or more "processing-oriented attributes" (properties) with an individual element of a document.

The purpose of property sheets is to permit end users, where so authorized, to exercise direct control over the appearance of elements in their documents. This control may be used for "fine tuning" of a document by introducing page breaks and similar small adjustments at places chosen by the user. Direct user control may also be used, with a relatively "permissive" document style, for creation of special effects in advertising copy and other types of documents whose appearance is not rigidly specified. By means of property sheets, Quill supports a spectrum of usage ranging from completely style-driven document types with no user-changeable properties to very flexible document types in which every property is controllable by the end user. These quite different usages, and many points in between, are supported with a common document syntax and user interface. The Designer's Workbench provides controls over the set of properties that are user-changeable, thus making user influence over document appearance a matter of policy rather than a hard-wired property of the formatting system.

Encoding property overrides in SGML Link Sets ensures that the property overrides are seen only by a specific process (the Quill formatter) and not by other applications which may be processing the document file.

It might be argued that direct user control over physical properties such as fonts and alignment is counter to the intent of SGML, which was designed for the description of "logical" documents. Of course, use of purely logical markup is very reasonable and useful, and the Quill Document Designer can force all markup to be purely logical simply by not authorizing any properties in the document to be overridden by users. Most generic-markup systems, however, provide some facilities for "final tuning" of the appearance of documents according to user discretion. It seems that automatic formatters have not yet reached such a state of perfection that users can resist "tinkering" with the final form of their pages. The mechanism provided by many markup-based systems for this "tuning" is insertion of low-level "control words" into the document by the user for such purposes as forcing line and page breaks or selecting a font. These low-level control words often use a syntax quite different from the logical markup; they do not conform to the hierarchical structure of the document; and they result in possibly conflicting instructions to

the formatter (e.g., a user instruction may call for a page break in the middle of an element that is defined to be atomic). As a result, low-level controls imbedded in an SGML document often lead to unexpected behavior and unhelpful error messages. The association of formatting properties with SGML elements, as in Quill, provides the "fine tuning" controls that are frequently needed, but does so in a way that never conflicts with the SGML syntax or semantics, and is under the direct control of an authorization mechanism.

Exposing property sheets to end users does, however, involve an important tradeoff. It is possible, by direct user control over document appearance, to introduce device dependencies (e.g., by naming a specific font) or style dependencies (e.g., by placing page breaks in places that are appropriate only for a specific style). These kinds of dependencies are the price that must be paid for the flexibility of "user tuning". If a document is intended to be used with more than one style or printer, it is important that "tuning" properties be kept to a minimum and used with full knowledge of their potential harmful effects.

3. Specifying Semantic Algorithms

It is a major objective of Quill to provide a facility for document designers to create new types of SGML elements and to define their semantics. Such a facility must meet two important requirements: it must be easy to use for specifying relatively simple elements, yet it must be capable of specifying complex semantics when necessary.

As described in the previous section, the behavior of many simple elements can be completely specified by means of a property sheet. The property sheet exposes all the variables that directly control the Quill formatting engine. Property sheets are capable of specifying element formatting rules such as "print my content in italic type, four points larger than the surrounding text", or "keep my content together on the same page, and print it in a monospaced font". Since property sheets allow the document designer to specify tag semantics by simple menu interactions, they meet the first Quill objective of providing an easy-to-use facility for defining simple elements.

The second objective, of providing a means for specifying complex semantics, requires an additional mechanism. As an example of a complex element, consider an index that is automatically generated by an "index" tag. The index generation process involves collecting information from

throughout the document, alphebetizing the index terms, eliminating duplicate page numbers, collapsing sequences of contiguous page numbers into ranges, etc. The index may be printed in one or more columns, with or without leader dots, and with various kinds of headings and subheadings. It is clearly infeasible to anticipate all possible kinds of index processing and list them as options on a property sheet—and an automatic index is only one example of an element with complex semantics.

The most powerful tool available for specifying semantic algorithms is a general-purpose programming language. For this reason, most existing systems that format documents based on an external style definition use some form of programming language in the style definition—for example, the ICEF2 system [Cham87a] uses a subset of Pascal for this purpose. However, semantic routines written in a programming language are usually based on a hidden assumption: that the document will be formatted from front to back, and that the semantic routines will be executed in order as the various elements are encountered in the document. This allows a semantic routine to share data with other routines, and with later activations of itself, by means of global variables. For example, a semantic routine for a "figure" element might increment the global variable containing the figure number by means of a statement such as `figno := figno + 1`. Hidden in this statement is the assumption that, when the routine is executed, the `figno` variable contains the correct value of the figure number for the current context in the document. In a WYSIWYG system such as Quill, however, any element in the document may be reformatted at any time due to user action. Thus a WYSIWYG system is a *reformatter* rather than a *formatter*, since its main job is to repair perturbations to a formatted document caused by the user. Figures anywhere in the document, for example, may be inserted, deleted, moved, or revised; therefore the semantic routine that defines the appearance and behavior of a figure must be executable at any time without any dependencies on the semantic routines of other elements.

We will define the term *WYSIWYG-able* in the following way: a semantic routine is *WYSIWYG-able* if it can be executed by itself at any time to reformat an individual element occurring anywhere in a document. To meet the objectives of Quill, it is necessary to define a semantic specification language that has the power and generality of a programming language, yet is WYSIWYG-able and also interfaces well with the property sheet approach to defining simple elements as discussed in the last section.

The language used for semantic specification in Quill is REXX [IBM84], a programming language implemented on several IBM machines. Each type of element in a document style is associated with both a property sheet and a semantic routine written in REXX. REXX provides the usual features of a high-level programming language such as looping, branching, and procedure calls. The key to the use of REXX as a specification language in Quill lies in the set of utility functions supported by Quill and made available to be called by the semantic routines of the various elements. These utility functions ensure that the semantic routines are WYSIWYG-able by recording all information needed for reformatting each element of the document in the "logical tree" data structure that is maintained during an editing session.

3.1. The OUTPUT Function

Perhaps the commonest utility function invoked by Quill semantic routines is the OUTPUT function, which places formatted text into the document. The OUTPUT function takes two arguments, as follows:

```
OUTPUT (properties, text)
```

If the `properties` argument is null (the usual case), the content of the `text` argument is formatted into the document according to the current element's property sheet (as overridden by the end user where permitted). The purpose of the `properties` argument is to allow the semantic routine to override one or more properties by specifying their names and values, such as `weight="bold"`. This type of override might be necessary if the element needs to print more than one piece of output having nonuniform properties; for example, an "abstract" element might print a heading in bold face, followed by the main part of the abstract in normal face. The semantic routine for such an element would consist of two OUTPUT functions, one with a property override to select bold face for the heading. An OUTPUT statement might also use a property override in order to set a property to some value that has been computed by program logic rather than simply selected in advance.[1]

1 Note that a property on a property sheet can be overridden either by an end user or by an OUTPUT function. These two kinds of override serve different purposes, and it is considered unlikely that a style definer would permit end-user overrides of a property that is also overridden by the semantic routine. If such a conflict should occur, the property value specified by the OUTPUT function becomes effective.

The `text` argument of the OUTPUT function specifies the text to be placed into the document. This text string may contain certain symbols beginning with the escape character "\", representing special functions such as tabs. The commonest such symbol is "\[]", which represents "the content of this element". Therefore, the semantic routine for most simple elements consists simply of the function `OUTPUT ('', '\[]');`, meaning "print my contents, using my property sheet". This one-line semantic routine is provided as the default routine for all elements unless otherwise specified, so the definer of a simple tag can interact with the property sheet only and need not be aware of the semantic routine.

When a new element is inserted into a document, its semantic routine is executed. Each OUTPUT statement in the routine places text into the formatted document, and links the text to a "content" node in the logical tree that records the effective properties of the text. Insertion of the new element may cause surrounding text to be "fixed" by reflowing it according to the properties recorded in its own "content" node(s). When the content of an element changes (e.g., a character is inserted or deleted by a user keystroke), the line containing the change, and as many adjacent lines as necessary, are "fixed" by reflowing their text according to their recorded properties. It is not necessary to re-execute a semantic routine on each keystroke.

When an element is deleted from the document by user action, its content is removed from the logical tree, and nearby elements are "fixed" by reflowing their text according to their recorded properties. When an element must be completely reformatted (e.g., it is moved into a new context or its SGML attributes are changed by user action), this is handled as a deletion followed by an insertion of the element (all the logical children of the element must be reformatted at the same time).

3.2. Functions for Sharing Data

As noted above, the OUTPUT function is WYSIWYG-able because it records the information it needs for re-execution in the logical tree. A more subtle requirement for a WYSIWYG-able function arises when a semantic routine needs to share data with another routine or with another activation of itself. As noted previously, this sharing of data cannot be accomplished simply by updating a global variable, since the order of execution of the various semantic routines is unpredictable. We will illustrate the requirement for shared data, and the WYSIWYG-able func-

tions provided by Quill for this purpose, by examining two example elements called <fig> and <figref>.

The <fig> element represents a figure that is to be printed together with its ordinal figure number (which must be generated automatically by the semantic routine). The <figref> element has an SGML attribute named "id"; its purpose is to generate a reference to the <fig> element with has a matching "id" attribute. For example, the SGML tag `<fig id="bicycle">` might print a figure with the generated caption "Figure 7"; the SGML tag `<figref id="bicycle">`, occurring either before or after the `<fig>` tag, would then generate the reference, "see Figure 7". As other figures are inserted, deleted, and moved, the number in the figure caption, and all references generated by `<figref>` tags, are automatically corrected.

The <fig> and <figref> elements illustrate two different kinds of shared data, which we will refer to by the names "local shared data" and "global shared data". Local shared data is represented by the propagation of the figure number from one <fig> element to the next. A local shared data item is characterized by a name (e.g. `figno`) and a value, but the value applies only to a local region of the document (e.g., between one <fig> element and the next). Global shared data is data that has validity throughout the document, such as the association of an ID with a particular figure number. A global shared data item is characterized by a qualified name that may include the ID attribute of an element (e.g., `figref.bicycle`) and a value. The value of a global shared data item may change over time, but at any given time it is valid throughout the document.

Quill provides two utility functions, LREAD and LWRITE, for handling local shared data, and two more functions, GREAD and GWRITE, for handling global shared data. Quill ensures that these data sharing functions are WYSIWYG-able by recording all shared data values in the logical tree, and by maintaining lists of all those elements in the document that read or write each shared data item.

When an element executes an LREAD for a given shared variable, Quill inserts the element into a linked list, called a "share list", consisting of all elements that read or write the given variable, maintained in their order of occurrence in the document. The LREAD function returns to the semantic routine the value recorded for the variable by the next-earlier LWRITE-type element in the share list. When an element executes an

LWRITE function for a given shared variable, it is linked into the share list in the appropriate position, and the value of the variable is recorded in the share list. If the value written by LWRITE is new or different from the value recorded by a previous execution of this element, it is necessary to re-execute other elements that depend on the value that has changed. The share list for the given data item is scanned in the forward direction, and all LREAD-type elements between the current LWRITE and the next LWRITE are marked for re-execution. Using spare cycles between user interactions, Quill will re-execute the semantic routines of the marked nodes, causing them to LREAD the new value of the shared data item. Thus, for example, when a new figure is added to the middle of a document, its semantic routine performs an LREAD to obtain the previous figure number, and also performs an LWRITE that causes the next figure element to be marked for re-execution. As each subsequent figure element is re-executed in turn using spare cycles, new figure numbers are propagated forward through the document.

Whenever an element is deleted whose logical tree node contains an LREAD or LWRITE notation, the deleted node is removed from any share lists in which it participates. In addition, if the deleted node had an LWRITE notation for some data item, all the following nodes on that share list are marked for re-execution, until the next LWRITE notation is encountered. Thus, for example, if a figure is deleted from a document, the following figures will be renumbered (one at a time).

The GREAD and GWRITE functions deal with global shared data, which propagates both forward and backward in the document. For each global shared data item, Quill records a value and an unordered list of elements that read the item. Whenever an element executes a GREAD function for a given shared variable, it is added to the share-list for that variable and the value of the variable is returned by GREAD. When an element executes a GWRITE for a given variable, the value of the variable is updated and all elements on the share-list for that variable are marked for re-execution. Using spare cycles between user interactions, Quill will re-execute the semantic routines of the marked nodes, causing them to GREAD the new value of the shared data item. Thus, for example, when the number of a figure changes, all references to that figure are marked for re-execution and updated using spare system cycles.

If an element that has performed a GWRITE is deleted from the document, the value of the corresponding variable becomes undefined,

and all the elements on its share-list are marked for re-execution. Thus, when a figure is deleted, all references to that figure throughout the document will be re-executed (and will become unresolved references).

The key to the WYSIWYG-able property of the Quill data-sharing functions is (1) each function records all its dependencies directly in the logical tree; and (2) whenever a function causes a data item to change, it finds all the nodes in the logical tree that are dependent on the changed item and marks them for re-execution. This basic approach is applied in Quill not only to data-sharing functions but also to utility functions that return other information such as a count of the current element's siblings or ancestors of a given type. A function that returns a count of left-brothers of a given type, for example, records in the logical tree-node a dependency on this count, and any subsequent change in the number of left-brothers causes this node to be marked for re-execution.

4. The Quill Designer's Workbench

We have seen that Quill defines the semantics of a given type of element by means of a property sheet and a semantic routine written in REXX. The combination of a property sheet and a semantic routine is called a *look*. In addition to its look, the definition of an element-type consists of two additional parts, which are required by the SGML standard: the *content model* and the *attribute definition list*. The content model of a given element specifies, by means of a regular expression, the permissible content of the element, including character data and other required or optional elements and their orderings. The attribute definition list specifies the attributes of the given element type, with their permissible values and default values. The content models and attribute definition lists for a given set of element-types can be formally expressed as an SGML *Document Type Definition*, which describes the syntax for a particular type of document; the corresponding semantics are described by the set of *looks* for the given elements.

The Quill Designer's Workbench is a program that allows a Document Designer to create a Design file that consists of content models, attribute definition lists, and looks for all the elements in a given type of document. The user of the Designer's Workbench may be an individual responsible for creating and enforcing style standards for a whole organization, or he may be simply an end user creating or revising a document style for his own personal use.

The output of the Designer's Workbench is a Design file that specifies the syntax and semantics of a particular type of document. The Design file serves as input to the Quill editor as it formats a document of the given type. However, it is desirable that a given *type* of document (as defined by a set of elements) be formattable in several different *styles.* For example, a certain type of document, called a "Techdoc", might consist of elements called paragraphs, headings, examples, lists, footnotes, etc. A particular Techdoc, however, could be formatted according to several different *styles*, such as those used by *Communications of the ACM*, IEEE *Computer*, and *IBM Journal of Research and Development.* The content of the document, and the set of tags and attributes used in its SGML markup, would be identical in all three cases; but the appearance of the document would be quite different as it is formatted under the control of three different *styles* named, perhaps, "ACM", "IEEE", and "IBM".

Thus, a Design file produced by the Designer's Workbench encapsulates one document *type* (e.g., "TECHDOC"), and one or more named *styles* (e.g., ACM, IEEE, IBM) that can be used with the given type. Each type of element in the Design file has a single content model, a single attribute definition list, and a look (property sheet and semantic routine) for each style. The Designer's Workbench presents all these objects to the user by means of a Design Matrix, as shown in Figure 2. All the elements of a given document type are listed on the left side of the matrix, and all the styles available for the given type of document are listed across the top of the matrix. Commands are provided for adding new elements and styles to the matrix. A notation in each cell of the matrix indicates whether that object is currently defined. By pointing to a cell in the matrix and pressing the mouse button, the Document Designer can invoke a specialized environment for creating or editing the selected object: a content model, attribute definition list, or look (property sheet and semantic routine).

If a Design file contains several styles, it may be desirable for some elements to share a common look across multiple styles. For example, a <paragraph> element might have the same look in two different styles, whereas a <heading> element might have different looks. The Quill Designer's Workbench allows looks to be shared across styles, thus saving space and eliminating redundancy. When the Document Designer is editing a look, he is presented with a Look Usage Table that indicates

Figure 2: The Design Matrix for a Document Type

DESIGN MATRIX	CONTENT	ATTRIB	ACM	IEEE	IBM
Techdoc	*	*	*	*	*
Title	*		*	*	*
Body	*		*	*	*
Paragraph	*	*	*	*	*
List	*	*		*	*
Item	*			*	*
Example	*	*	*	*	
Figure	*	*	*	*	*

how looks are shared by the various styles for the given element. Commands are provided for creating new looks and for reassigning looks from one style to another.

5. Summary

The Quill project at IBM Research is exploring ways to combine the flexibility and ease of use of WYSIWYG systems with the formatting power and standard enforcement abilities of generic-markup systems. The resulting system will present a WYSIWYG user interface, but will format documents under the control of an external Design file that provides varying degrees of user control over appearance, according to the policy of the Document Designer. A tool called the Designer's Workbench is provided that enables the Document Designer to specify the syntax and semantics of each element in a given type of document, and to control those aspects of each element's appearance that can be overridden by end users. The appearance of each element is controlled by a "look" that consists of a property sheet and an (often defaulted) semantic routine. The semantic routines, needed only for complex tags, are written in the REXX language, using a set of functions that are specially designed for WYSIWYG processing. In order to be suitable for WYSIWYG processing, semantic routines must be executable at any time, in any order;

therefore they must record their dependencies in a shared data structure, and the semantic routine of each element must be automatically re-executed whenever a change occurs in one of its dependencies.

6. References

[Cham87a] Chamberlin, D.D. (1987). Document Convergence in an Interactive Formatting System, *IBM Journal of Research and Development*, **31**, 1, pp. 58-72.

[Cham87b] Chamberlin, D.D., and Goldfarb, C.F. (1987). Graphic Applications of the Standard Generalized Markup Language (SGML), *Computers and Graphics*, **11**, 4.

[Cham87c] Chamberlin, D.D., Hasselmeier, H.F., Luniewski, A.W., Paris, D.P., Wade, B.W., and Zolliker, M.L. (1987). Quill: An Extensible System for Editing Documents of Mixed Type. In *Proceedings of the 21st Hawaii International Conference on System Sciences.* Washington, DC: IEEE Computer Society Press.

[Good85] Goodfellow, M.J. (1985). Whim, the Window Handler and Input Manager. In *Proceedings of the First International Conference on Computer Workstations*, pp. 12-21. Washington, DC: IEEE Computer Society Press.

[IBM84] *Virtual Machine/System Product: Interpreter User's Guide.* IBM Publication No. SC24-5238.

[INT86] *Interleaf Workstation Publishing Software User's Guide.* Interleaf, Inc.

[ISO86] *International Standard ISO 8879: Information Processing - Text and Office Systems - Standard Generalized Markup Language (SGML).* International Organization for Standardization.

[Reid80] Reid, B.K. *Scribe: A Document Specification Language and Its Compiler.* Ph.D. Dissertation and Technical Report CMU-CS-81-100, Carnegie-Mellon University, Pittsburgh, PA (1980). Scribe is a registered trademark of Unilogic Ltd., Pittsburgh, PA.

[Schn83] Schneiderman, B. Direct Manipulation: A Step Beyond Programming Languages, *Computer*, **16**, 8, pp. 57-69.